Expressive and Extensible Parameter Passing for Distributed Object Systems

ELI TILEVICH and SRIRAM GOPAL, Virginia Tech

In modern distributed object systems, reference parameters to a remote method are passed according to their runtime type. This design choice limits the expressiveness, readability, and maintainability of distributed applications. Further, to extend the built-in set of parameter passing semantics of a distributed object system, the programmer has to understand and modify the underlying middleware implementation. To address these design shortcomings, this article presents (i) a declarative and extensible approach to remote parameter passing that decouples parameter passing semantics from parameter types, and (ii) a plugin-based framework, *DeXteR*, which enables the programmer to extend the built-in set of remote parameter passing semantics, without having to understand or modify the underlying middleware implementation. DeXteR treats remote parameter passing as a distributed cross-cutting concern and uses aspect-oriented and generative techniques. DeXteR enables the implementation of different parameter passing semantics as reusable application-level plugins, applicable to application, system, and third-party library classes. The expressiveness, flexibility, and extensibility of the approach is validated by adding several non-trivial remote parameter passing semantics (i.e., copy-restore, lazy, streaming) to Java Remote Method Invocation (RMI) as DeXteR plugins.

Categories and Subject Descriptors: D.1.2 [**Programming Techniques**]: Automatic Programming—*Program synthesis*; D.1.3 [**Programming Techniques**]: —*Distributed Programming*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Frameworks*

General Terms: Languages, Design

Additional Key Words and Phrases: Extensible Middleware, Metadata, Aspect Oriented Programming (AOP), Generative Programming, Declarative Programming

1. INTRODUCTION

One of the most widely-used paradigms for constructing distributed systems is Remote Procedure Call (RPC) [Birrell and Nelson 1984]. RPC leverages the ubiquity of procedure calls in programming languages and makes remote calls work like local calls. RPC is also straightforward to implement—most programming languages can express remote calls by simply defining an appropriate library.

To support the construction of distributed object systems [The Object Management Group (OMG) 1997; Wollrath et al. 1996], RPC has been extended to Remote Method Invocation (RMI), which enables transparent object distribution.

This is a corrected and expanded version of a paper [Gopal et al. 2008] presented at the 9th ACM/IFIP/USENIX Middleware Conference.

Corresponding Author's Address: Eli Tilevich, Dept. of Computer Science, Virginia Tech, Blacksburg, VA, USA. Email: tilevich@cs.vt.edu.

Although RMI-based distributed object systems aim at making remote methods indistinguishable from local methods, parameter passing is one implementation facet that is handled specially. Parameter passing in local calls can be implemented efficiently because the caller and the callee share the same address space. When distribution takes place, however, efficiently maintaining the local parameter passing semantics across different address spaces becomes impossible without special hardware support. As a result, distributed object systems offer different parameter passing semantic for remote calls (e.g., *call-by-copy*, *call-by-remote-reference*, *call-by-copy-restore*). Furthermore, to designate how a particular parameter should be passed to a remote method, the programmer has to use types. Specifically, the runtime type of a remote parameter determines the semantics by which it will be passed.

Types are among the most commonly explored research topics in programming languages. In addition to presenting stimulating research challenges, type systems provide practical means for creating more robust programs. By ensuring that program fragments compute a specific set of values, a type system helps avoid undesirable runtime behaviors. In his book "Types and Programming Languages," Pierce points out how type systems benefit practical software development by detecting errors, providing useful abstractions, documenting program behavior, ensuring language safety, and improving efficiency [Pierce 2002].

Despite all of these benefits of type systems, the main argument of this article is that *types are ill-suited as a mechanism to express the semantics for passing parameters to remote methods.* We argue that type-based parameter passing is a poorly-designed programming abstraction that negatively affects the expressiveness, readability, and maintainability of distributed applications.

Mainstream programming languages such as C, C++, and C# express the choice of local parameter passing mechanisms through method declarations with special tokens rather than types. For example, by default objects in C++ are passed by *value*, but inserting the & token after the type of a parameter signals the by *reference* mechanism. Why cannot distributed object systems adhere to a similar declarative paradigm for remote method calls, albeit properly designed for distributed communication?

To that end, we present *declaration-based parameter passing* for remote methods. While Java always uses the by *value* semantics for local calls, distributed communication requires a richer set of semantics to ensure good performance and to increase flexibility. With our model, the declaration of a remote method includes a passing mechanism for each parameter, expressed using Java 5 annotations. We demonstrate how decoupling parameter passing from parameter types increases expressiveness, improves readability, and eases maintainability. We also show that IDL-based distributed object systems such as CORBA [Group 1998b] and DCOM [Brown and Kindel 1998] with their in, out, and inout parameter modes stop short of a fully declarative parameter passing model and are not extensible.

Recognizing that many existing distributed applications are built upon a type-based model, we present a technique for transforming a type-based remote parameter passing model to use a declaration-based one. Our technique transforms parameter passing functionality transparently, without any changes to the under-

lying distributed object system implementation, ensuring cross-platform compatibility and ease of adoption. With Java RMI as our example domain, we combine aspect-oriented and generative techniques to retrofit its parameter passing functionality. Our approach is equally applicable to application classes, system classes, and third-party libraries.

In addition, declaration-based remote parameter passing simplifies adding new semantics to an existing distributed object system. We present an extensible plug-in-based framework, through which third-party vendors or in-house expert programmers can seamlessly extend a built-in set of remote parameter passing semantics. Our framework allows such extension in the application space, without modifying the JVM or its runtime classes. As a validation, we used our framework to extend the set of available parameter passing semantics of RMI with several non-trivial state-of-the-art semantics, introduced earlier in the literature both by us [Tilevich and Smaragdakis 2008] and others [Line et al. 2008; Yang et al. 2006; Eberhard and Tripathi 2001].

One of the implemented semantics optimizes our own prior algorithm for *copy-restore* [Tilevich and Smaragdakis 2008]. While the original implementation is inefficient in high-latency, low-bandwidth network environments, the optimized version of the *copy-restore* algorithm—*copy-restore with delta*—efficiently identifies and encodes the changes made by the server to a copy-restore parameter.

Contributions and Roadmap

This article makes the following novel contributions:

—A clear exposition of the shortcomings of type-based parameter passing in distributed object systems, including CORBA, Java RMI, and .NET Remoting.

—An alternative declarative parameter passing approach that offers design and implementation advantages.

—An extensible framework to retrofit RMI applications with declaration-based parameter passing and to extend the built-in set of semantics.

—An enhanced *copy-restore* mode of remote parameter passing, offering performance advantages for low bandwidth, high latency networks.

The rest of this article is structured as follows. Section 2 demonstrates the shortcomings of type-based parameter passing using a bioinformatics application. Section 3 presents DeXteR, our extensible framework. Section 4 describes how we used DeXteR to add several non-trivial parameter passing semantics to RMI. Section 5 discusses the advantages and constraints of our approach. Section 6 discusses related work. Section 7 outlines future work directions, and Section 8 presents concluding remarks.

2. MOTIVATING EXAMPLE

Organizations have hundreds of workstations connected into local area networks (LANs) that stay unused for hours at a time. Consider leveraging these idle computing resources for distributed scientific computation. Specifically, we would like

to set up an ad-hoc grid that will use the idle workstations to solve bioinformatics problems. The ad-hoc grid will coordinate the constituent workstations to align, mutate, and cross DNA sequences, thereby solving a computationally intensive problem in parallel.

Each workstation has a standard Java Virtual Machine (JVM) installed, and the LAN environment makes Java RMI a viable distribution middleware choice. As a distributed object model for Java, RMI simplifies distributed programming by exposing remote method invocations through a convenient programming model. In addition, the synchronous communication model of Java RMI aligns well with the reliable networking environment of a LAN.

The bioinformatics application follows a simple Master-Worker architecture, with classes Sequence, SequenceDB, and Worker representing a DNA sequence, a collection of sequences, and a worker process, respectively. Class Worker implements three computationally-intensive methods: align, cross, and mutate.

```
1 interface WorkerInterface {
2   void align(SequenceDB allSeqs, SequenceDB candidates,
3              Sequence toMatch);
4   Sequence cross(Sequence s1, Sequence s2);
5   void mutate(SequenceDB seqs);
6 }
7
8 class Worker implements WorkerInterface { ... }
```

The align method iterates over a collection of candidate sequences (candidates), adding to the global collection (allSeqs) those sequences that satisfy a minimum alignment threshold. The cross method simulates the crossing over of two sequences (e.g., during mating) and returns the offspring sequence. Finally, the mutate method simulates the effect of a gene therapy treatment on a collection of sequences, thereby mutating the contents of every sequence in the collection.

Consider using Java RMI to distribute this application on an ad-hoc grid, so that multiple workers could solve the problem in parallel. To ensure good performance, we need to select the most appropriate semantics for passing parameters to remote methods. However, as we demonstrate next, despite its Java-like programming model, RMI uses a different remote parameter passing model that is *type-based*. That is, the runtime type of a reference parameter determines the semantics by which RMI passes it to remote methods. We argue that this parameter passing model has serious shortcomings, with negative consequences for the development, understanding, and maintenance of distributed applications.

Method align takes two parameters of type SequenceDB: allseqs and candidates. allseqs is an extremely large global collection that is being updated by multiple workers. We, therefore, need to pass it by *remote-reference*. candidates, on the other hand, is a much smaller collection that is being used only by a single worker. We, therefore, need to pass it by *copy*, so that its contents can be examined and compared efficiently. To pass parameters by *remote-reference* and by *copy*, the RMI programmer has to create subclasses implementing marker interfaces Remote and Serializable, respectively. As a consequence, method align's signature must be changed as well. Passing allSeqs by *remote-reference* requires the type of allSeqs

to become a remote interface. Finally, examining the declaration of the remote method align would give no indication about how its parameters are passed, forcing the programmer to examine the declaration of each parameter's type. In addition, in the absence of detailed source code comments, the programmer has no choice but to examine the logic of the entire slice [De Lucia et al. 1996] of a distributed application that can affect the runtime type of a remote parameter.

Method mutate mutates the contents of every sequence in its seqs parameter. Since the client needs to use the mutated sequences, the changes have to be reflected in the client's JVM. The situation at hand renders passing by *remote-reference* ineffective, since the large number of remote callbacks is likely to incur a significant performance overhead. One approach is to pass seqs by *copy-restore*, a semantics which efficiently approximates *remote-reference* under certain assumptions [Tilevich and Smaragdakis 2008].

Because Java RMI does not natively support *copy-restore*, one could use a custom implementation provided either by a third-party vendor or an in-house expert programmer. Mainstream middleware, however, does not provide programming facilities for such extensions. Thus, adding the new semantics would require that a programmer understand the RMI implementation in sufficient detail to be able to add the new functionality by hand either by manipulating bytecode directly or using a tool as AspectJ.

Finally, consider the task of maintaining the resulting ad-hoc grid distributed application. Assume that SequenceDB is a remote type in one version of the application, such that RMI will pass all instances SequenceDB by *remote-reference*. However, if a maintenance task necessitates passing some instance of SequenceDB using different semantics, the SequenceDB type would have to be changed. Nevertheless, if SequenceDB is part of a third-party library, it may not be subject to modification by the maintenance programmer.

To summarize, type-based parameter passing is detrimental for the development and maintenance of distributed applications. Relying on types also makes it difficult to extend a distributed object system with additional remote passing semantics.

3. DECLARATION-BASED PARAMETER PASSING WITH *DEXTER*

This section discusses the design and implementation of **DeXteR** (**D**eclarative **Exte**nsible **R**emote Parameter Passing)—our framework for declarative and extensible remote parameter passing. We begin by demonstrating how the bioinformatics application presented above can be distributed with ease if declaration-based parameter passing is used. We then provide a general overview of DeXteR—its API, implementation insights, and design assumptions.

3.1 Bioinformatics Example Revisited

Using DeXteR, the programmer can express remote parameter passing semantics by annotating remote method declarations with the intended semantics. A DeXteR parameter passing plug-in provides both the annotation and the implementation for a given semantics. Using annotations rather than types to express different parameter passing semantics reduces the amount of changes the programmer must make to distribute a centralized application. For instance, a distributed version of the bioinformatics application from Section 2 can be expressed using DeXteR as

follows.

```
1  interface WorkerInterface extends Remote
2  {
3     void align(@RemoteRef SequenceDB matchingSeqs,
4                @Copy SequenceDB candidates,
5                @Copy Sequence toMatch)
6                         throws RemoteException;
7
8     @Copy Sequence cross(@Copy Sequence s1,
9                          @Copy Sequence s2)
10                           throws RemoteException;
11
12    void mutate(@CopyRestore SequenceDB seqs)
13                           throws RemoteException;
14 }
```

Since remote parameter passing annotations are part of a remote method's signature, they must appear in both the method declaration in the remote interface and the method definitions in all remote classes implementing the interface. This requirement ensures that the client is informed about how remote parameters will be passed, and it also allows for safe polymorphism (i.e., the same remote interface may have multiple remote classes implementing it). This requirement however, must not impose any additional burden on the programmer, as a modern IDE such as Eclipse [Foundation 2007], NetBeans [Oracle Corporation 2010], or Visual Studio [Microsoft Corporation 2007] can be configured to reproduce these annotations when providing method stub implementations for remote interfaces.

3.2 Framework Overview

DeXteR implements declaration-based remote parameter passing on top of standard Java RMI, without modifying its implementation. DeXteR uses a plug-in based architecture and treats remote parameter passing as a distributed cross-cutting concern. Each parameter passing semantics is an independent plugin component.

DeXteR uses the Interceptor Pattern [Schmidt et al. 2000] to expose the invocation context explicitly on the client and the server sites. The Interceptor pattern captures techniques for extending the functionality of a complex system at specific interception points. While Interceptors have been used in several prior systems [Narasimhan et al. 1999; Fleury and Reverbel 2003] to introduce orthogonal cross-cutting concerns such as logging and security, the novelty of our approach lies in employing Interceptors to transform and enhance the core functionality of a distributed object system, its remote parameter passing semantics.

Figure 1 depicts the overall translation strategy employed by DeXteR. The rank-and-file (i.e., application) programmer annotates an RMI application with the desired remote parameter passing semantics. The annotations processor takes the application source code as input, and extracts the programmer's intent. The extracted information parameterizes the source code generator, which encompasses the framework-specific code generator and the plugin-specific code generators. The framework-specific code generator synthesizes the code for the client and the server interceptors using aspects. The plugin-specific code generators synthesize the code

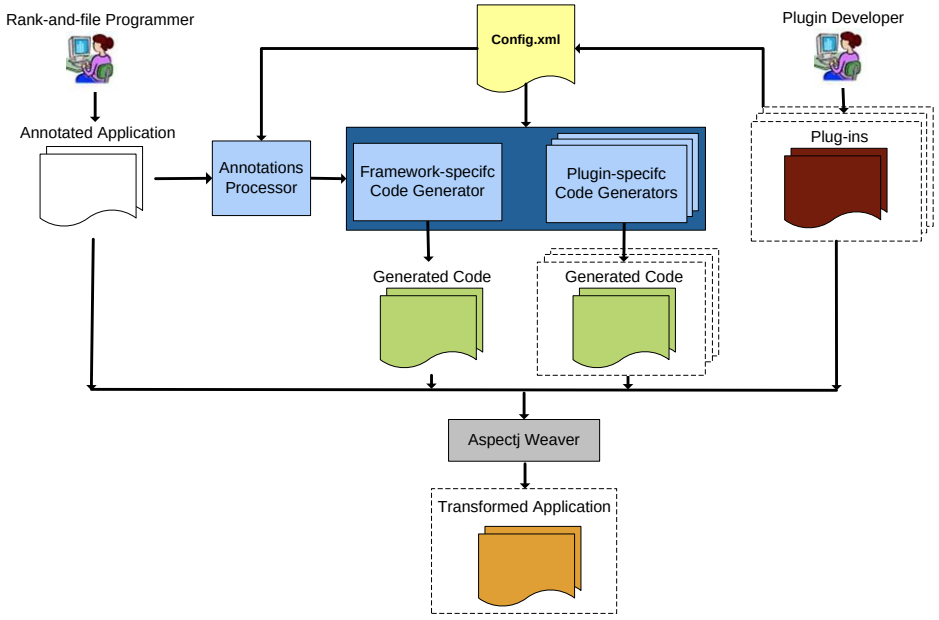

Fig. 1. Development and deployment process using DeXteR.

pertaining to the translation strategy for supporting a specific parameter passing semantics. DeXteR compiles the generated code into bytecode, and the resulting application uses standard Java RMI, only with a small AspectJ runtime library as an extra dependency. The generated aspects are weaved into the respective classes at load-time, thereby redirecting the invocation to the framework interceptors at both the local and the remote sites.

3.3 Framework API

DeXteR provides extension points for parameter passing plugins in the form of the IGenerator interface and the InterceptionPoint interface. Developing a new plugin involves implementing the InterceptionPoint interface and the optional IGenerator interface, identifying the interception points of interest, providing the functionality at these interception points, and registering the plugin with the framework.

```
1 interface IGenerator { // Plugin−specific code generator
2     void generate(AnnotationInfo info);
3 }
```

The IGenerator interface forms the compile-time part of a plugin. At compile-time, DeXteR exposes the annotation information extracted from the RMI application to the respective parameter passing plugins. Plugins can use this information to generate code, which can then be used at run-time for implementing a specific parameter passing strategy.

```
1 interface InterceptionPoint {
2     // Interception points on client−side
```

```
3       ... argsBeforeClientCall (...);
4       ... customArgsBeforeClientCall (...);
5       ... retAfterClientCall (...);
6       ... customRetAfterClientCall (...);
7
8       // Interception points on server-side
9       ... argsBeforeServerCall (...);
10      ... customArgsBeforeServerCall (...);
11      ... retAfterServerCall (...);
12      ... customRetAfterServerCall (...);
13  }
```

The InterceptionPoint interface forms the run-time part of a plugin. It exposes the invocation context of a remote call at different points of its control-flow on both the client and server sites. DeXteR exposes to a plugin only the invocation context pertaining to the corresponding parameter passing annotation. For example, plugin X obtains access only to those remote parameters annotated with annotation X. DeXteR enables plugins to modify the original invocation arguments. Plugins can thus modify the invocation arguments using the code generated at compile-time. In addition, DeXteR enables sending custom information between the client- and the server-side plugins. This custom information is simply piggy-backed to the original invocation context.

3.4 Implementation Details

DeXteR intercepts the invocation of remote methods by combining aspect-oriented and generative programming techniques. Specifically, DeXteR adds methods to RMI remote interface, stub, and server implementation classes by means of AspectJ. Following the Proxy pattern, the intercepted methods delegate to the added methods, thus interposing the logic required to support various remote parameter passing strategies. The aspect code itself is automatically generated for each distributed application.

3.4.1 *Compile-time.* For each remote method, DeXteR generates AspectJ code that injects a wrapper method into the remote interface and the server implementation using *inter-type declarations*, which enable introducing new members. In addition, DeXteR pointcuts on the execution of that method in the stub (i.e., implemented as a dynamic proxy) to provide a wrapper. This is accomplished by providing an *around* advice, which runs in place of a specific execution point. All the AspectJ code that provides the interception functionality is automatically generated at compile time, based on the remote method's signature.

3.4.2 *Load-time.* The generated aspects are weaved into the stub (i.e., dynamic proxy) on the client side, and the remote interface and server implementation on the server side when these classes are loaded into the virtual machine.

3.4.3 *Runtime.* At runtime, the flow of a remote call is intercepted to invoke the plugins with the annotated parameters, and the modified set of parameters is obtained. The intercepted invocation on the client site is then redirected to the added extra method on the server. The added server method reverses the process, invoking the parameter passing style plugins with the modified set of parameters

provided by their client-side peers. The resulting parameters are used to make the invocation on the actual server method. A similar process occurs when the call returns, in order to support different passing styles for return values as well.

3.5 Applicability Assumptions

In terms of applicability of DeXteR, we assume a distributed system written in a modern mainstream language that (1) features language-integrated metadata facilities (e.g., Java 5 annotations), (2) loads code at runtime, and (3) has a dynamic aspect oriented extension that can weave in extra functionality. Consequently, our approach would not be directly applicable to legacy systems written in older languages (e.g., C and C++) that do not feature the facilities described above. Even though one could try to adapt various facets of our approach to make it work with legacy systems written in these languages, it is not clear how complex or even feasible such adaptations would be.

In modern software development, the languages that satisfy our assumptions are commonly described as *managed languages*, such as Java and C#. Because managed languages reduce the complexity of software construction by providing portability, type safety, and automated memory management, they have become dominant in enterprise software development—a Gartner report predicts that in 2010, as much as 80% of new software will be written in C# or Java [Flen and Linden 2005]. Because managed languages and environments are widely used in the construction of distributed enterprise systems, our approach can benefit a high and growing percentage of real word systems.

4. SUPPORTING PARAMETER PASSING SEMANTICS

This section describes how several non-trivial parameter passing semantics, previously proposed in the research literature [Tilevich and Smaragdakis 2008; Line et al. 2008; Yang et al. 2006; Eberhard and Tripathi 2001], can be implemented as DeXteR plugins.

To demonstrate the power and expressiveness of our approach, we chose the semantics that have very different implementation requirements. While the lazy and streaming semantics require proxies for the parameters' classes, copy-restore requires passing extra information between the client and the server. Despite the contrasting nature of these semantics, we were able to encapsulate all their implementation logic inside their respective plugins and easily deploy them using DeXteR. In terms of the programming effort involved, it took in the order of a couple of weeks for an M.S. student without prior knowledge of the RMI internals to implement each of the semantics described in this section.

4.1 Lazy Semantics

Lazy parameter passing [Line et al. 2008], also known as *lazy pass-by-value*, is intended for asynchronous distributed environments, especially in P2P applications. It works by passing the object initially by reference and then transferring it by value either upon first use (*implicitly lazy*) or at a point dictated by the application (*explicitly lazy*). More precisely, lazy parameter passing defines *if and when exactly an object is to be passed by value.*

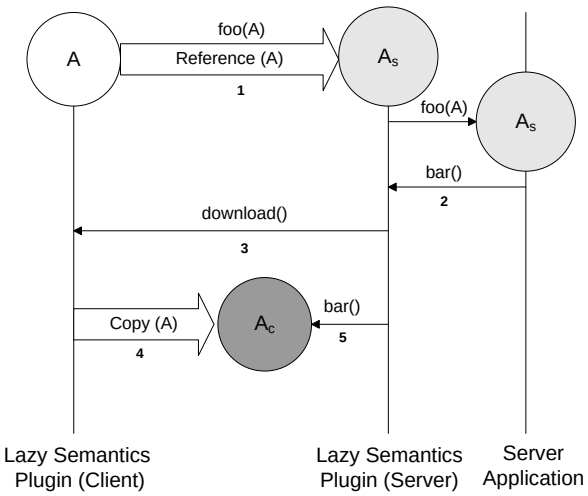

Fig. 2. Lazy semantics plugin interaction diagram
(A: Serializable Object; A_s: Stub of A; A_c: Copy of A; (1) A is passed from client to server; (2) Server invokes foo() on stub A_s; (3) Server plugin calls download() on client plugin; (4) Client plugin sends a copy of A, A_c; (5) Server plugin calls foo() on A_c.)

The translation strategy for passing reference objects by *lazy* semantics involves using the plugin-specific code generator. As our aim is to decouple parameter types from the semantics by which they are passed, to pass a parameter of type A by *lazy* semantics does not require defining any special interface nor A implementing one. Instead, the plugin-specific code generator generates a Remote interface, declaring all the accessible methods of A. To make our approach applicable for passing both application and system classes, we deliberately avoid making any changes to the parameter's class A. Instead, we use a delegating dynamic proxy (e.g., A_DynamicProxy) for the generated Remote interface (e.g., AIface) and generate a corresponding server-side proxy (e.g., A_ServerProxy) that is type-compatible with the parameter's class A.

As is common with proxy replacements for remote communication [Eugster 2006], all the direct field accesses of the *remote-reference* parameter on the server are replaced with accessor and mutator methods.[1]

In order to enable obtaining a copy of the remote parameter (at some point in execution), the plugin inserts an additional method download() in the generated remote interface AIface, the client proxy A_DynamicProxy and the server proxy A_ServerProxy.

```
1
2 class A {
3   public void foo() {...}
4 }
5
```

[1]Replacing direct fields accesses with methods has become such a common transformation that AspectJ [Kiczales et al. 2001] provides special fields access pointcuts (i.e., set, get) to support it.

```
6  // Generated remote interface
7  interface AIface extends Remote {
8      public void foo() throws RemoteException;
9      public A download() throws RemoteException;
10 }
11
12 // Generated client proxy
13 class A_DynamicProxy implements AIface {
14     private A remoteParameter;
15
16     public A download() {
17         // serialize remoteParameter
18     }
19
20     public void foo() throws RemoteException { ... }
21 }
22
23 // Generated server proxy
24 class A_ServerProxy extends A {
25     private A a;
26     private AIface stub;
27
28     public A_ServerProxy(AIface stub) {
29         this.stub = stub;
30     }
31
32     synchronized void download() {
33         // Obtain a copy of the remote parameter
34         a = stub.download();
35     }
36
37     public void foo() {
38         // Dereference the stub
39         stub.download();
40         // Invoke the method on the copy
41         a.foo();
42     }
43 }
```

Any invocation made on the parameter (i.e., server proxy) by the server results in a call to its download method, if a local copy of the parameter is not yet available. The download method of the server proxy replays the call to the download method of the enclosed client proxy with the aim of obtaining a copy of the remote parameter.

The client proxy needs to serialize a copy of the parameter. However, passing a remote object (i.e., one that implements a Remote interface) by *copy* presents a unique challenge, as type-based parameter passing mechanisms are deeply entangled with Java RMI. The RMI runtime replaces the object with its stub, effectively forcing pass by *remote-reference*. The plugin-generated code overrides this default functionality of Java RMI by marshaling a given remote object into a `byte` array using *serialization*. This technique effectively "hides" the remote object, as the

RMI runtime transfers `byte` arrays without inspecting or modifying their content. The "hidden" remote object can then be extracted from the array on the server-side by the server proxy. Once the copy is obtained, all subsequent invocations made on the parameter (i.e., server proxy) are delegated to the local copy of the parameter.

Thus, passing an object of type A as a parameter to a remote method will result in the client-side plugin replacing it with its type-incompatible stub. The server-side plugin wraps this type-incompatible stub into the generated server-side proxy that is type-compatible with the original remote object.

We note that a subset of the strategies described above is used for supporting declarative parameter passing for the native RMI semantics of *copy* and *remote-reference*.

4.2 Copy Restore Semantics

A semantics with a different set of implementation requirements than that of *lazy* parameter passing is the *copy-restore* semantics. It copies a parameter to the server and then restores the changes to the original object in place (i.e., preserving client-side aliases). A fully general implementation of *copy-restore* must work correctly for parameters that are linked data structures (e.g., lists, trees, etc.), some of which may share structure.

Such an implementation must offer identical semantics to *remote-reference* in the important case of single-threaded clients and stateless servers (i.e., when the server cannot maintain state reachable from the arguments of a call after the end of the call). It is the responsibility of the programmer to ensure that these preconditions are met before specifying *copy-restore* for a given remote parameter. Using *copy-restore* when these preconditions are unmet constitutes a programming error.

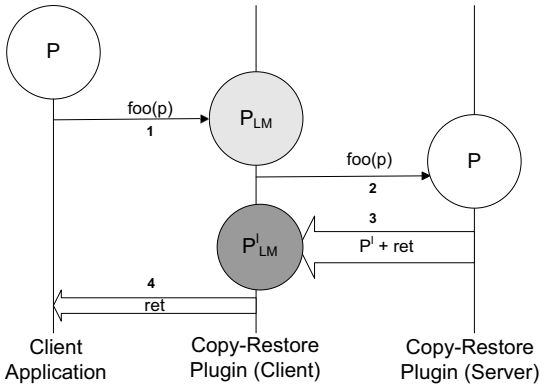

Fig. 3. Copy-restore semantics plugin interaction diagram
(P: Set of parameters passed to foo; P_{LM}: Linear map of parameters; P^l: Modified parameters (restorable data); *ret*: values returned by the invocation; $P^1{}_{LM}$: Modified linear map; (1) The client invokes method foo() passing parameter p; (2) The client-side plugin constructs a linear map P_{LM} and calls the original foo(p); (3) Server-side plugin invokes foo and returns modified parameters P^l and the return value *ret*; (4) Changes restored and the return value *ret* is passed to the client.)

Implementing the *copy-restore* semantics involves tracing the invocation arguments and restoring the changes made by the server after the call. The task is simplified by the well-defined hook points provided by the framework. The *copy-restore* plugin obtains a copy of the parameter A and creates a linear map of all objects reachable from the parameter on both the client and the server sites prior to the invocation. The invocation then resumes and the server mutates the parameter during the call. Once the call completes, the server-side plugin needs to send back to its client-side peer the changes made by the server represented as a linear map of all objects reachable from the parameter. This is accomplished using the custom information passing facility provided by the framework. The client-side plugin obtains the linear map from its server-side peer, compares it with the linear map it holds, and restores the changes to the original parameter A in the client's JVM.

4.3 Copy Restore With Delta Semantics

For single-threaded clients and stateless servers, *copy-restore* makes remote calls indistinguishable from local calls as far as parameter passing is concerned [Tilevich and Smaragdakis 2008]. However, in a low bandwidth high latency networking environment, such as in a typical wireless network, the reference *copy-restore* implementation may be inefficient. The potential inefficiency lies in the restore step of the algorithm, which always sends back to the client an entire object graph of the parameter, no matter how much of it has been modified by the server. To optimize the implementation of *copy-restore* for low bandwidth, high latency networks, the restore step can send back a "delta" structure by encoding the differences between the original and the modified objects. The necessity for such an optimized *copy-restore* implementation again presents a compelling case for extensibility and flexibility in remote parameter passing.

The following pseudo-code describes our optimized *copy-restore* algorithm, which we term *copy restore with delta*:

(1) Create and keep a linear map of all the objects transitively reachable from the parameter.
(2) On the server, again create a linear map, Lmap1, of all the objects transitively reachable from the parameter.
(3) Deep copy Lmap1 to an isomorphic linear map Lmap2.
(4) Execute the remote method, modifying the parameter and Lmap1, but not Lmap2.
(5) Return Lmap1 back to the client; when serializing Lmap1, encode the changes to the parameter by comparing with Lmap2 as follows:
    (a) Write as is each changed existing object or a newly added object.
    (b) Write its numeric index in Lmap1 for each unchanged existing object.
(6) On the client, apply the encoded changes and use the client-side linear map to retrieve the original (unchanged) old objects at the specified indexes.

Creating Linear Map. A linear map of objects transitively reachable from a reference argument is obtained by recording each encountered object during serial-

ization. In order not to interfere with garbage collection, all linear maps use weak references.

Calculating Delta. The algorithm encodes the delta information efficiently using a handle structure shown below.

```
1 class Handle{
2     int id;
3     ArrayList<Long> chId;
4     ArrayList<Long> chScript;
5     ArrayList<Object> chObject;
6 }
```

The identifier id refers to the position of an object in the client site linear map. The change indicator chId identifies the modified member fields using a bit level encoding. chScript contains the changes to be replayed on the old object. For a primitive field, its index simply contains the new value, whereas for an object field, its index points to chObject, which contains the modified references.

Restoring Changes. For each unserialized handle on the client, the corresponding old object is obtained from the client's linear map using the handle identifier id. The handle is replaced with the old object, and the changes encoded in the handle are replayed on it. Following the change restoration, garbage collection reclaims the unused references.

As a concrete example of our algorithm, consider a simple binary tree, t, of integers. Every node in the tree has three fields: data, left, and right. A subset of the tree is aliased by non-tree pointers alias1 and alias2. Consider a remote method such as the one shown below, to which tree t is passed as a parameter.

```
1 void alterTree (Tree tree) {
2     tree.left.data = 0;
3     tree.right.data = 9;
4     tree.right.right.data = 8;
5     tree.left = null;
6     Tree temp = new Tree (2, tree.right.right, null);
7     tree.right.right = null;
8     tree.right = temp;
9 }
```

Figure 4 shows the sequence of steps involved in passing tree t by *copy restore with delta* and restoring the changes made by the remote method alterTree to the original tree.

We measured the performance gains of our algorithm over the original copy-restore by conducting a series of micro-benchmarks, varying the size of a binary tree and the amount of changes performed by the server. The benchmarks were run on Pentium 2.GHz (dual core) machines with 2 GB RAM, running Sun JVM version 1.6.0 on an 802.11b wireless LAN. Figures 5 and 6 show the percentage of performance gain of copy-restore with delta over copy-restore. Overall, our experiments indicate that the performance gain is directly proportional to the size of the object graph and is inversely proportional to the amount of changes made to the object graph by the server.

By providing flexibility in parameter passing, DeXteR enables programmers to

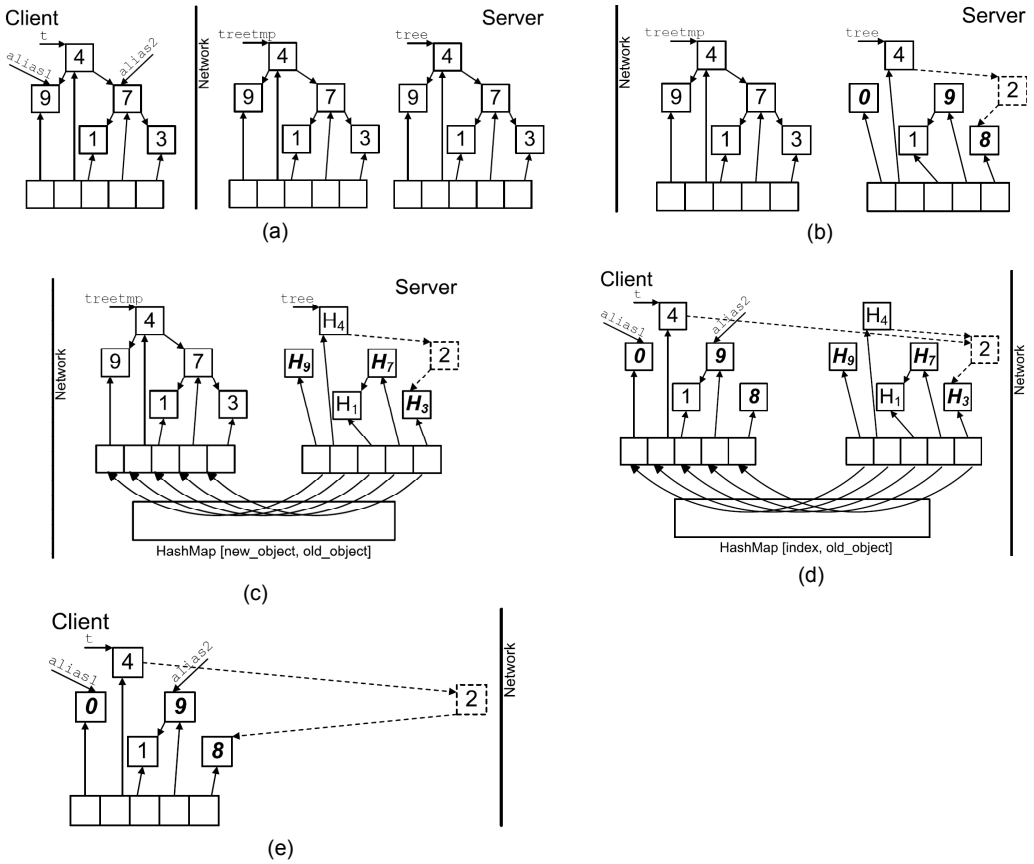

Fig. 4. *Copy-restore with delta* algorithm by example (a) State after step 3. (b) State after step 4. The remote procedure modified the parameter. (c) State during step 5. Copy the modified objects (even those no longer reachable through `tree`) back to the client; compute the delta script for modified objects using a hash map. (d) State during step 6. Replace the handles with the original old objects; replay the delta script to reflect changes. (e) State of the client side object after step 6.

use not only different semantics, but also different variations of the same semantics as required by the nature of the application. For instance, within the same application one can pass parameters by regular *copy-restore* to a method operating on small parameters and by *copy-restore with delta* to a method operating on larger ones.

4.4 Streaming Semantics

Passing objects by *streaming* is useful when parameters or return types are large objects. It involves buffering large objects in the background, without blocking the call. As streaming is similar to the lazy semantics, so are their respective implementation strategies. The key difference between the two semantics lies in the

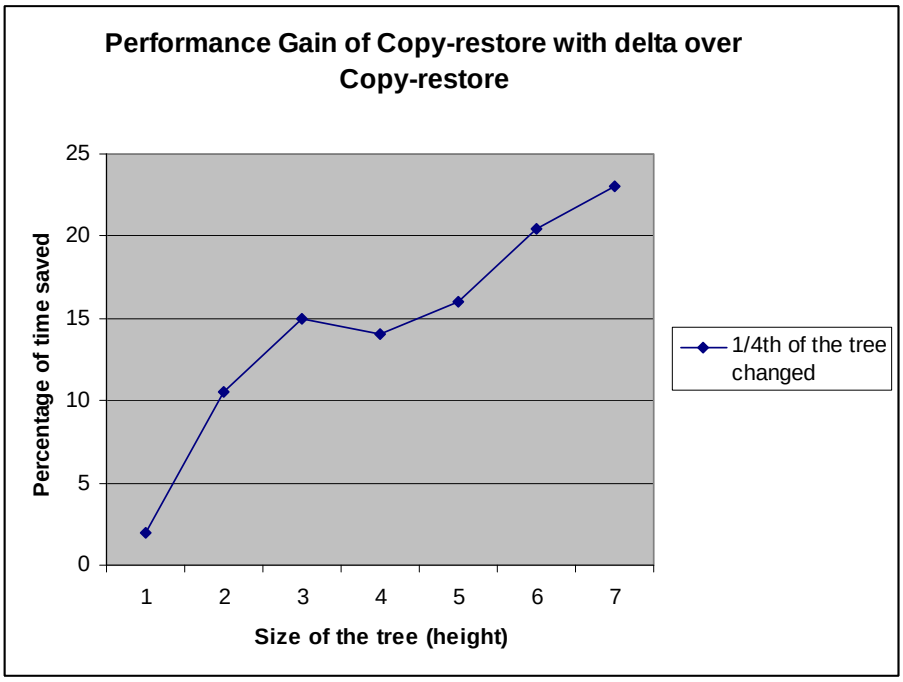

Fig. 5. The advantages of saving bandwidth via "copy-restore with delta" (the server changes 1/4th of the binary tree's nodes)

way the copy of an object is obtained. Having described how the lazy semantics is implemented for passing parameters, the discussion below focuses on implementing streaming for return values.

Our strategy for supporting streaming involves transmitting an object initially by *reference* by employing a pair of proxies (A_DynamicProxy and A_ClientProxy). We use the plugin-specific code generator to generate the proxies and the remote interface during compile time. Returning an object of type A will result in the server-side plugin replacing it with a type-incompatible proxy A_DynamicProxy. The client-side plugin wraps this type-incompatible proxy into a stub A_ClientProxy that is type-compatible with the return type of the remote method. Prior to returning the wrapped object to the client, the client-side streaming plugin obtains a weak reference to it, so that its referent would not be prevented from being garbage collected. Then the client code spawns a thread whose aim is to obtain a local copy of the returned object. The spawned thread invokes the download method on the type-incompatible proxy A_DynamicProxy enclosed within the type-compatible stub A_ClientProxy instance. The download method returns a copy of A, which is populated within the A_ClientProxy instance by the client-side plugin using the weak reference it holds.

Future invocations made by the client on the return type are handled at the client end, as soon as a copy of the entire object has been streamed. If not, the

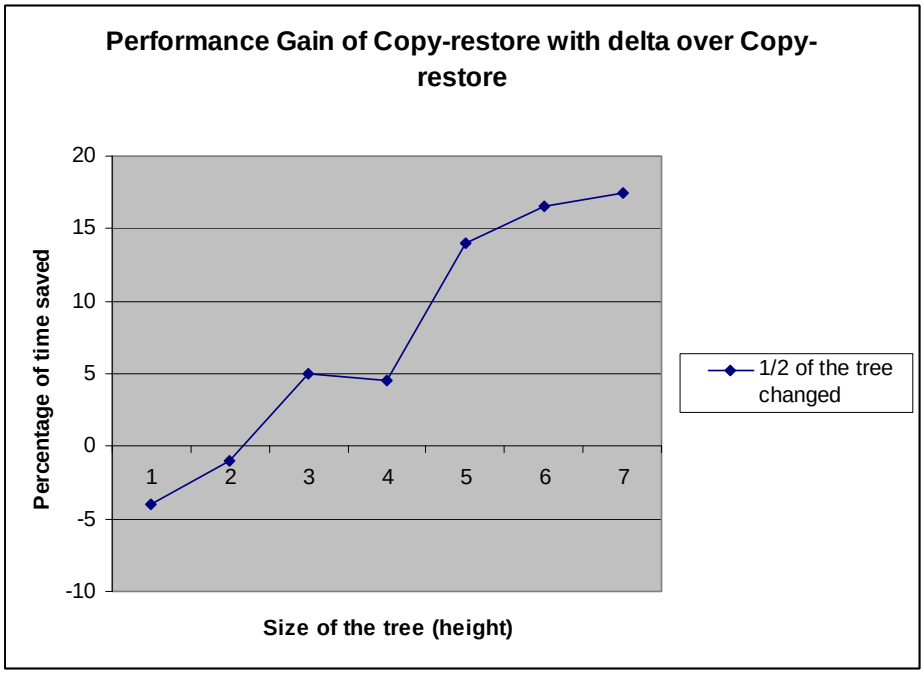

Fig. 6. The advantages of saving bandwidth via "copy-restore with delta" (the server changes 1/2 of the binary tree's nodes)

invocations are delegated to the server using the member *remote-reference*.

```
1  // Generated client proxy
2  class A_ClientProxy extends A
3  {
4      // streamed copy of the remote object
5      private A a;
6      // type−incompatible stub
7      private AIface stub;
8      // streaming completion indicator
9      private boolean isBuffered;
10
11     public A_ClientProxy(AIface stub) {
12         this.a = null;
13         this.stub = stub;
14         this.isBuffered = false;
15     }
16
17     public void deref() {
18         // obtain a copy of the remote parameter
19         a = stub.deref();
20     }
21
```

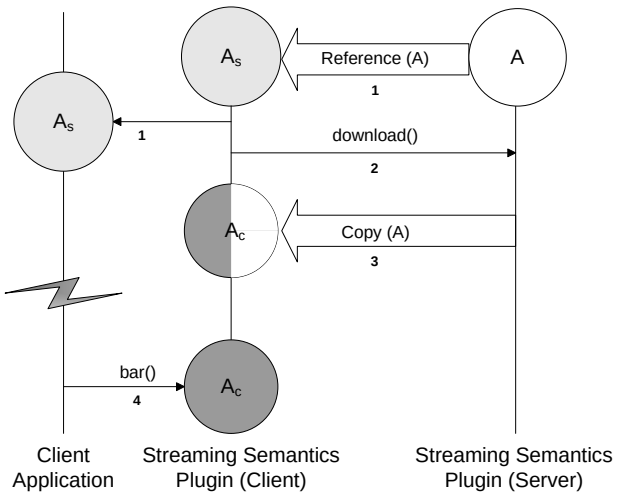

Fig. 7. Streaming semantics plugin interaction diagram
(A: Serializable Object; A_s: Stub of A; A_c: Copy of A; (1) A is returned from server to client; (2) Client plugin spawns a thread and calls download() on server plugin; (3) Server plugin sends a copy of A, A_c and the client plugin starts buffering it; (4) Client calls foo() on the buffered A_c.)

```
22    public void setBufferedStatus(){
23        isBuffered = true;
24    }
25
26    public boolean isBuffered(){
27        return isBuffered ;
28    }
29
30    public void bar() {
31        if( isBuffered ()) {
32            // invoke the method on the copy
33            a.bar();
34        }
35        else {
36            // invoke the method on the remote object
37            stub.bar();
38        }
39    }
40 }
```

In a distributed systems context, the term *streaming* is often interpreted as allowing some of an object's state to be used before the entire state was received. The implementation described above is a special kind of `future` for a remote object. While the streamed object has not been completely received, all the calls to it are forwarded to the object's origin. One can view this implementation as a form of passing by *asynchronous copy*. Thus, our use of the term "streaming" refers to what transpires at the implementation level, rather than to what the term means

in the traditional distributed systems context from the end-user's perspective.

4.5 Caching Semantics

In order to validate the expressive power of DeXteR further, we chose to implement a variant of *parameter substitution* a.k.a *caching* that follows a simple caching strategy and consistency policy. Reference [Eberhard and Tripathi 2001] describes other strategies for parameter substitution based on different consistency guarantees.

Parameter passing by *caching* can be used when unchanged resources/parameters are sent frequently from the client to the server or vice-versa. It involves saving a copy of the state of parameter objects on the receiving node and using them for subsequent invocations without requiring a retransmission. *Caching* effects are more pronounced when the cost of caching computations is subsumed by the savings in bandwidth.

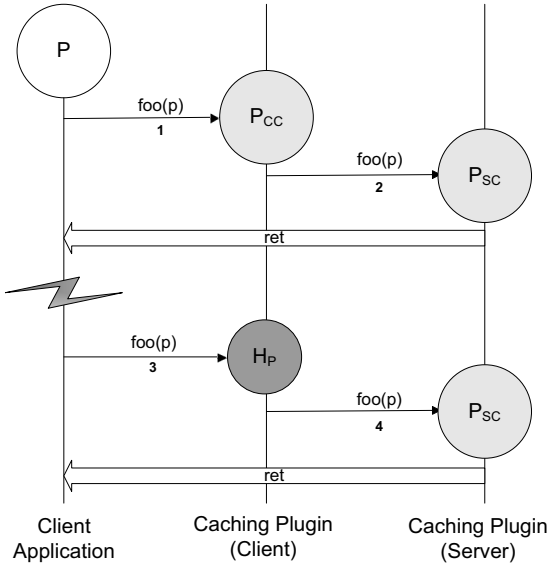

Fig. 8. Caching semantics plugin interaction diagram
(P: Set of parameters passed to foo; P_{CC}: Parameter in client cache; P_{SC}: Parameter in server cache; H_P: Handle for parameter P; (1) The client invokes method foo() passing parameter P; (2) The client-side and the server-side plugins cache P as P_{CC} and P_{SC} respectively before invoking the original foo(p); (3) The client makes a subsequent invocation of foo() with the same parameter P; (4) The client-side plugin replaces P with handle H_P and the server-side plugin retrieves P_{SC} corresponding to H_P and invokes foo() with P_{SC}.)

When a parameter is transmitted for the first time, the client-side caching plugin stores a copy of the parameter in its local cache prior to serializing a copy of it to the server using the pass by *copy* strategy outlined earlier. On obtaining the parameter object, the server-side caching plugin stores a copy of it in its local cache prior to providing the remote method with the parameter object. Since the plugin

caches a deep copy of the parameter object, mutating the parameter object does not affect the state of the cached object.

Subsequent invocations involving the same parameter object state result in the client-side caching plugin substituting the original parameter object with an object identifier and sending it to the server. The server-side plugin reverses the process: it uses the identifier sent by its client-side peer to retrieve the object state from its local cache and uses this cached object as the parameter for the remote method.

4.6 Future Semantics

DeXteR offers the advantage of supporting a wide variety of remote parameter passing semantics through a uniform API. By decoupling parameter passing semantics from parameter types, DeXteR flexibly supports new parameter passing semantics as well as new optimization strategies. Developments in hardware and software designs are likely to lead to the creation of new parameter passing semantics and optimization mechanisms. These mechanisms will leverage the new designs, but may be too experimental to be included in the implementation of a standard middleware system. DeXteR will allow the integration and use of these novel mechanisms at the application layer, without changing the underlying middleware. As a particular example, consider the introduction of massive parallelism into mainstream processors. Multiple cores will require the use of explicit parallelism to improve performance. Some facets of parameter passing are computation-intensive and can benefit from parallel processing. One can imagine, for instance, how marshaling could be performed in parallel, in which parts of an object graph are serialized/deserialized by different cores.

5. DISCUSSION

This section discusses some of the advantages of the DeXteR framework as well as some of the constraints imposed by our design.

5.1 Design Advantages

Expressing remote parameter passing choices as a part of a method declaration has several advantages over a type-based system. Specifically, a declaration-based approach increases expressiveness/encapsulation, improves readability, and eases maintainability. To further illustrate the advantages of our declaration-based framework, we compare and contrast our approach with that of Java RMI.

5.1.1 *Expressiveness/Encapsulation.* In Java RMI, all instances of the same type are passed identically, which restricts expressiveness. To select a different semantics, the programmer can create a subclass that implements the required marker interface. Thus, potentially the same remote method can use different passing semantics for the same parameter, as determined by the parameter's runtime type. In essence, the remote method's programmer cannot enforce by which semantics its parameters are passed, as it can change between call sites. A remote method encapsulates a certain behavior; a call site determining the method's parameter passing semantics may violate the encapsulation principle.

By contrast, our approach does not require any new subclasses to be created or any changes to be made to the original method's signature. The simple declara-

tive style of our annotations makes enforcement of the parameter passing policies straightforward. With our approach, a remote method fully encapsulates the semantics by which its parameters are passed.

5.1.2 *Readability.* Examining the declaration of a remote method does not reveal any details about how its parameters are passed. Furthermore, the programmer has to exhaustively examine each call site to discover which exact subclasses for each remote parameter have been used. Forcing the programmer to examine all the program's parameter types and call sites reduces readability and hinders program understanding. By contrast, our approach provides a single point of reference that explicitly informs the programmer how remote parameters are passed.

5.1.3 *Maintainability.* An existing class may have to be modified to implement an interface before its instances can be passed as parameters to a remote method. This complicates maintainability as, in the case of third-party libraries, source code may be difficult or even impossible to modify. By contrast, our approach enables the maintenance programmer to modify the semantics by simply specifying a different parameter passing annotation.

5.1.4 *Extensibility.* Even if the *copy-restore* semantics is natively supported in the next version of Java RMI, including new optimization mechanisms such as using *copy-restore with delta* would still require modifying the underlying RMI implementation of both the client and the server. By contrast, our approach supports extending the native remote parameter passing semantics at the application-level, requiring no modifications to the underlying middleware.

5.1.5 *Reusability.* DeXteR also enables providing the parameter passing semantics as plugin libraries. Application programmers thus can obtain third-party plugins and automatically enhance their own RMI applications with the new parameter passing semantics.

5.1.6 *Efficiency.* Another advantage of our approach is its efficiency. All the DeXteR transformations do not cause any additional overhead in using objects of type A, until they are passed using a particular semantics in an RMI call. This requires that one know exactly when an object of type A is used in this capacity. The insight that makes it possible to detect such cases precisely is that the program execution flow must enter an RMI stub (dynamic proxy) for a remote call to occur.

Once a parameter is passed to a remote method, however, DeXteR then introduces several levels of proxy indirection, and we had to ensure that this indirection overhead is not unreasonable. To measure the overhead imposed by DeXteR, we conducted a series of micro-benchmarks that compared the performance of pass by *copy* and pass by *remote-reference* semantics implemented as DeXteR plugins with that of their native RMI counterparts. The results represent the average of running each benchmark 1,000 times on a Pentium D 3.GHz (dual core) machine with 2GB of RAM, running Sun JVM version 1.6.0. By warming the JVM, we ensured that the measured programs had been dynamically compiled before measurements. To distill the pure overhead, we ran the benchmarks on a single machine. In the presence of network communication and added latency, the overhead incurred by the additional levels of local indirection would be dominated. Therefore, the results do

not unfairly benefit our approach.

Since pass by *copy* predominantly involves the cost of object serialization, its micro-benchmark involved measuring the execution times for a varying object size. Figure 9 presents the performance comparison of the two implementations of pass by *copy*.

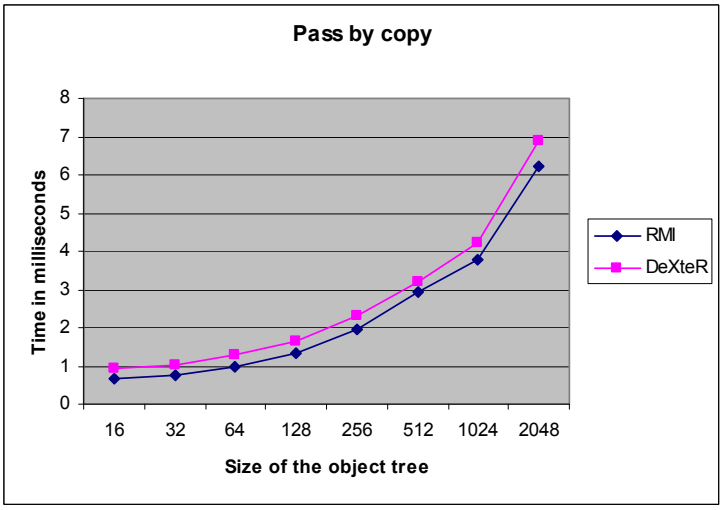

Fig. 9. Pass by copy benchmark.

In lieu of support for type-compatible dynamic proxies for classes in Java, the DeXteR implementation emulates this functionality using a type-incompatible client-side dynamic proxy and a type-compatible server-side wrapper proxy. Thus, this emulated functionality introduces two new levels of indirection compared to the standard Java RMI implementation of pass by *remote-reference*. The purpose of passing a parameter by *remote-reference* is to enable the server to invoke methods on that parameter as part of the remote method's logic. Since these invocations will be propagated back to the client, they are called *remote callbacks*. Figure 10 presents the performance comparison between the two implementations of pass by *remote-reference*, for a varying number of remote callbacks.

As the latency of a remote call is orders of magnitude greater than that of a local call, one can expect that the overhead incurred by the additional local calls added to the remote call by DeXteR would be insignificant. Indeed, the resulting overhead never exceeds a few percentage points of the total latency of a remote call executed on a single machine. Thus, the performance measurements above confirm the feasibility of our approach: the small overhead incurred by DeXteR is well offset by its software engineering benefits.

5.2 Design Constraints

Achieving the afore-mentioned advantages without changing the Java language required constraining our design in the following ways.

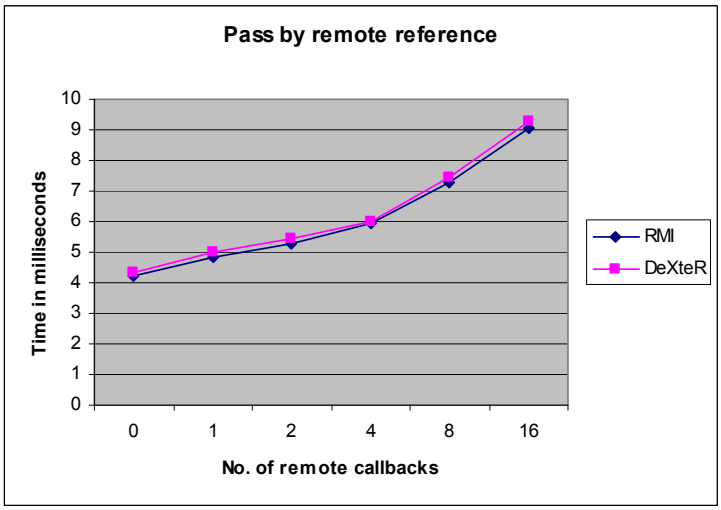

Fig. 10. Pass by remote-reference benchmark.

Classes Analyzed	Total	Classes with Public Fields	Total Public Fields
User-Accessible Classes	2732	57	123
GUI Classes	913	15	65
Exception Classes	364	33	34
RMI Classes	58	22	22
Java Bean Classes	56	3	3

Table I.
Analysis of Java 6 JDK's public member fields (some overlap exists due to Exception classes spanning multiple packages).

First, array objects are always passed by *copy* though the array elements could be passed using any desired semantics. While this is a limitation of our system, it is still nonetheless an improvement over standard RMI, which also passes array objects by *copy*, but passes array elements based on their runtime type.

Second, passing **final** classes (not extending UnicastRemoteObject) by *remote-reference* would entail either removing their **final** specifier or performing a sophisticated global replacement with an isomorphic type [Tilevich and Smaragdakis 2002]. This requirement stems from our translation strategy's need to create a proxy subclass for *remote-reference* parameters, an impossibility for **final** classes. Since heavy transformations would clash with our design goal of simplicity, our approach issues a compile-time error to an attempt to pass an instance of a **final** class by *remote-reference*. Again, this limitation is also shared by standard RMI.

Finally, since our approach does not modify standard Java classes, it is not possible to support direct member field access for instances of system classes passed by *remote-reference*. While this is a conceptual problem, an analysis of the Java 6 library shown in Table 1 indicates that this is not a practical problem. For our purposes, we analyzed the java.* and javax.* classes, as they are typically the only

ones used by application developers. As the table demonstrates, approximately 1% of classes contain non-final member fields. However, the vast majority of these classes are either GUI or sound components, SQL driver descriptors, RMI internal classes, or exception classes, and as such, are unlikely to be passed by *remote-reference*. Additionally, the classes in java.beans.* provide getter methods for their public fields, thereby not requiring direct access. The conclusion of our analysis is that only one (java.io.StreamTokenizer) of more than 5,500 analyzed classes could potentially pose a problem, with two public member fields not accessible by getter methods.

6. RELATED WORK

The body of research literature on distributed object systems and separation of concerns is extremely large and diverse. The following discusses only closely-related state of the art.

6.1 Separation of Concerns

Several language-based and middleware-based approaches address the challenges in modeling cross-cutting concerns.

Proxies and Wrappers [Souder and Mancoridis 1999] introduce late bound cross-cutting features, though in an application-specific manner.

Aspect Oriented Programming (AOP) [Kiczales et al. 1997] is a methodology for modularizing cross-cutting concerns. Several prior AOP approaches aim at improving various properties of middleware systems, with the primary focus on modularization [Zhang and Jacobsen 2003; Eichberg and Mezini 2004].

Java Aspect Components (JAC) [Pawlak et al. 2004] and DJCutter [Nishizawa et al. 2004] support distributed AOP. The JAC framework makes it possible to add and remove an advice dynamically. DJCutter extends AspectJ with *remote pointcuts*, a special language construct for developing distributed systems. DeXteR could use these approaches as an alternative to AspectJ.

A closely related work is the DADO [Wohlstadter et al. 2003] system for programming cross-cutting features in distributed heterogeneous systems. Similar to DeXteR, DADO uses hook-based extension patterns. It employs a pair of user-defined adaplets, explicitly modeled using IDL for expressing the cross-cutting behavior. To accommodate heterogeneity, DADO employs a custom DAIDL (an IDL extension) compiler, runtime software extensions, and tool support for dynamically retrofitting services into CORBA applications. DADO uses the Portable Interceptor approach for triggering the advice for cross-cutting concerns, which do not modify invocation arguments and return types. Thus, using DADO to change built-in remote parameter passing semantics would not eliminate the need for binary transformations and code generation.

The SPOON [Pawlak 2005] framework provides a program transformation tool that takes advantage of Java 5 annotations to define and parameterize user-defined transformations. Using compile-time reflection, SPOON enables annotation driven AOP with pure Java. Base programs can thus be annotated to define how and where the aspects are weaved. SPOON can be used as an alternative to AspectJ in our implementation.

6.2 Remote Parameter Passing

Multi-language distributed object systems, such as CORBA [Group 1998b], DCOM [Brown and Kindel 1998], use an Interface Definition Language (IDL) to express how parameters are passed to remote methods. Each parameter in a remote method signature is associated with keywords in, out, and inout designating the different passing options. The IDL specification is translated into a conventional programming language such as C, C++ or Java. Traditional RPC systems, thus, have separated the IDL and the target language mappings for flexibility reasons.

The design of Java RMI, however, no longer distinguishes between a language-independent IDL specification and a mapping to a given implementation language. Specifically, in RMI, Java interfaces have supplanted IDL specifications. Despite the simplicity advantages of this design, it lacks flexibility when it comes to remote parameter passing. Our framework, DeXteR, addresses this particular shortcoming of the RMI design. In fact, DeXteR goes beyond some IDL-based approaches that can be limited in flexibility if the language binding cannot be adapted as necessary [Ford et al. 1994; 1995].

In addition, mainstream IDL implementations do not completely decouple parameter passing semantics from parameter types. When an IDL interface is mapped to a concrete language, the generated implementation may still rely on a type-based parameter passing model of the target language. As an example, CORBA in mapping IDL to Java [Group 2003], an IDL *valuetype* maps to a Serializable class, which is always passed by *copy*. Conversely, an IDL *interface* maps to a Remote class, which is always passed by *remote-reference*. Additionally, even if we constrain parameters to *valuetypes* only, the mapped implementation will generate different types based on the keyword modifiers present [Group 1998a]. Thus, remote parameter passing in mainstream IDL-based distributed object systems is neither fully declarative, nor it is extensible.

.NET Remoting [Obermeyer and Hawkins 2001] for C# also follows a mixed approach to remote parameter passing. It supports the parameter-passing keywords *out* and *ref*. However, the *ref* keyword designates pass by *value-result* in remote calls rather than the standard pass by *reference* in local calls. This difference in passing semantics may lead to the introduction of subtle inconsistencies when adapting a centralized program for distributed execution. Furthermore, in the absence of any optional parameter passing keywords, a reference object is passed based on the parameter type. While this approach shares the limitations of Java RMI, *remote-reference* proxies are type-compatible stubs, which provide full access to the remote object's fields. Therefore, while the parameter passing model of .NET Remoting contains some declarative elements, it has shortcomings and is not extensible.

Adaptive parameter passing [Lopes 1997] optimizes RPC marshaling by sending a subset of an object's state graph, specified using a domain specific language. Adaptive parameter passing can be thought of as a mechanism for defining custom distributed parameter passing semantics on a per-object basis. Our approach can be extended to allow a comparable level of flexibility; the programmer can be given greater control by means of annotation attributes that can be set individually to customize the passing semantics for each remote parameter.

The Quality Objects (QuO) framework [Schantz et al. 2002] provides adaptive quality of service (QoS) capabilities as standard middleware services. As their runtime environments change, applications built using QuO can adapt their execution to meet the required QoS requirements. Among the functional interface-specific adaptations offered by QuO is the ability to change the characteristics of method parameters. Our approach similarly offers flexible and extensible remote parameter passing as a standard middleware service.

Doorastha [Dahm 2000] represents a closely related piece of work on increasing the expressiveness of distributed object systems. It aims at providing distribution transparency by enabling the programmer to annotate a centralized application with distribution tags such as *globalizable* and *by-refvalue*, and using a specialized compiler for processing the annotations to provide fine-grained control over the parameter passing functionality. While influenced by the design of Doorastha, our approach differs in the following ways. First, Doorastha does not completely decouple parameter passing from the parameter types, as it requires annotating classes of remote parameters with the desired passing style. Unannotated remote parameters are passed based on their type. Second, Doorastha does not support extending the default set of parameter passing modes. Finally, Doorastha requires a specialized compiler for processing the annotations. While Doorastha demonstrates the feasibility of many of our approach's features, we believe our work is the first to present a comprehensive argument and design for a purely declarative and extensible approach to remote parameter passing.

The Opentalk communication layer [Cincom Systems Inc. 2002] is a set of frameworks and components that provide a rich and extensible environment for development, deployment, maintenance, and monitoring of distributed Smalltalk applications. As other distributed object systems, it supports pass by *value* and pass by *reference*. By default, all immediate objects (nil , **true**, **false**, Characters and SmallIntegers), Magnitudes, ByteStrings, ByteSymbols, and some collections are passed by *value*. Parameters that are complex objects are exported to the distributed runtime automatically and passed by reference. However, the framework enables the programmer to override this default behavior by forcing objects to be passed by *value* or by *reference* using special keywords asPassedByValue and asPassedByRef. This approach, however, does not follow a fully declarative style, as the default behavior is still type-based. Furthermore, the native parameter passing modes are not extensible to include other semantics.

6.3 Web Services

A *web service* is a remote invocation that uses the world wide web as the transport protocol. The Simple Object Access Protocol (SOAP) can be used to send and receive structured XML documents [Box et al. 2002]. With respect to parameter passing, SOAP web services only allow defined record-like types and certain primitive types to be transferred.

An alternative to SOAP web services is *representation state transfer* (REST) [Fielding and Taylor 2000], an architectural model that resembles a very abstract remote shell command. A REST request is a URL with a path and form parameters, commonly interpreted as an object address and method parameters. The output can be any valid hypertext media such as HTML or images. The simple

request model of REST transfers form parameters as plain text.

Thus, our approach to declarative and extensible remote parameter passing is unlikely to be applicable to the modern implementations of web services.

7. FUTURE WORK

A promising future work direction is to develop a declaration-based distributed object system for an emerging object-oriented language such as *Ruby* [Thomas and Hunt 2001] or *Scala* [Odersky et al. 2004]. It would be interesting to explore how advanced language features of Ruby such as built-in aspects, closures, and co-routines can be utilized in the implementation. Despite the exploratory nature and the presence of advanced features in the Ruby language, *DRuby* [Seki 2007], its distributed object system, does not significantly differ from Java RMI.

Scala currently does not even have a distributed object system, but due to its interoperability with Java, Scala programs can use Java RMI. Scala language is extensible and supports Java-style annotations. Some of Scala extensibility features include extensible types, and control structures, operator overloading, etc. We would like to explore how these advanced features can aid it the development a new generation distributed object system for Scala. This system can leverage all the improvements that had been proposed for Java RMI, including a declarative and extensible distributed parameter passing model.

8. CONCLUSIONS

This article has exposed the shortcomings of type-based remote parameter passing models in distributed object systems. To overcome these shortcomings, we presented declaration-based parameter passing as a better alternative to type-based parameter passing. We have also presented an argument in favor of treating parameter parameter passing as a distributed cross-cutting concern, separate from the core functionality of a distributed object system. Based on this principle, we have created an extensible framework for declaration-based parameter passing and described how multiple non-trivial semantics can be efficiently implemented on top of a type-based parameter passing model with ease using our extensible framework. Our experiences have shown that the framework provides a powerful distributed programming platform and a convenient experimentation facility for research in distributed object systems.

Availability

DeXteR can be downloaded from `http://research.cs.vt.edu/vtspaces/dexter`.

REFERENCES

Birrell, A. and Nelson, B. 1984. Implementing remote procedure calls. *ACM Transactions on Computer Systems (TOCS) 2,* 1, 39–59.

Box, D., Ehnebuske, D., Kakivaya, G., Layman, A., Mendelsohn, N., Nielsen, H. F., Thatte, S., and Winer, D. 2002. Simple object access protocol (soap) version 1.1.

Brown, N. and Kindel, C. 1998. Distributed Component Object Model Protocol–DCOM/1.0. Redmond, WA, 1996.

Cincom Systems Inc. 2002. Opentalk Communication Layer Developer's Guide. `http://www.cincom.com/downloads/pdf/OpentalkDevGuide.pdf`.

Dahm, M. 2000. Doorasthaa step towards distribution transparency. In *Proceedings of the Net. Object Days 2000*.

De Lucia, A., Fasolino, A., and Munro, M. 1996. Understanding function behaviours through program slicing. In 4^{th} *IEEE Workshop on Program Comprehension*. 9–18.

Eberhard, J. and Tripathi, A. 2001. Efficient Object Caching for Distributed Java RMI Applications. *Lecture Notes In Computer Science 2218*, 15–35.

Eichberg, M. and Mezini, M. 2004. Alice: Modularization of Middleware using Aspect-Oriented Programming. *Software Engineering and Middleware (SEM) 2004*.

Eugster, P. 2006. Uniform proxies for java. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*. ACM Press, New York, NY, USA, 139–152.

Fielding, R. T. and Taylor, R. N. 2000. Principled design of the modern web architecture. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*. ACM, New York, NY, USA, 407–416.

Flen, J. and Linden, A. 2005. Gartners hype cycle special report. Tech. rep., Gartner Research. `www.gartner.com`.

Fleury, M. and Reverbel, F. 2003. The JBoss Extensible Server. *International Middleware Conference*.

Ford, B., Hibler, M., and Lepreau, J. 1994. Separating presentation from interface in RPC and IDLs.

Ford, B., Hibler, M., and Lepreau, J. 1995. Using annotated interface definitions to optimize RPC. *ACM SIGOPS Operating Systems Review 29,* 5.

Foundation, T. E. 2007. Eclipse - an open development platform. `http://www.eclipse.org`.

Gopal, S., Tansey, W., Kannan, G. C., and Tilevich, E. 2008. DeXteR – an extensible framework for declarative parameter passing in distributed object systems. In *ACM/IFIP/USENIX* 9^{th} *Middleware Conference*.

Group, O. M. 1998a. Objects By Value. document orbos/98-01-18. Framingham, MA.

Group, O. M. 1998b. The Common Object Request Broker: Architecture and Specification. Framingham, MA.

Group, O. M. 2003. IDL to Java Language Mapping Specification. Framingham, MA.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. 2001. An overview of AspectJ. *Lecture Notes in Computer Science 2072,* 327-355, 110.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J., and Irwin, J. 1997. Aspect-Oriented Programming. *ECOOP'97–object-oriented Programming: 11th European Conference, Jyväskylä, Finland, June 9-13, 1997: Proceedings*.

Line, C., Jayaram, K. R., and Eugster, P. 2008. Lazy argument passing in java rmi. In *PPPJ '08: Proceedings of the 6th international symposium on Principles and practice of programming in Java*. ACM, New York, NY, USA, 127–136.

Lopes, C. 1997. D: A language framework for distributed programming. Ph.D. thesis, Northeastern University.

Microsoft Corporation. 2007. Visual Studio 2005. `msdn.microsoft.com/vstudio`.

Narasimhan, P., Moser, L., and Melliar-Smith, P. 1999. Using interceptors to enhance CORBA. *Computer 32,* 7, 62–68.

Nishizawa, M., Chiba, S., and Tatsubori, M. 2004. Remote pointcut: a language construct for distributed AOP. *Proceedings of the 3rd international conference on Aspect-oriented software development*, 7–15.

Obermeyer, P. and Hawkins, J. 2001. Microsoft .NET Remoting: A Technical Overview. *MSDN Library*.

Odersky, M., Altherr, P., Cremet, V., Emir, B., Maneth, S., Micheloud, S., Mihaylov, N., Schinz, M., Stenman, E., and Zenger, M. 2004. An overview of the Scala programming language. *LAMP-EPFL*.

Oracle Corporation. 2010. NetBeans IDE. `http://www.netbeans.org`.

Pawlak, R. 2005. Spoon: annotation-driven program transformation—the AOP case. *Proceedings of the 1st workshop on Aspect oriented middleware development*.

Pawlak, R., Seinturier, L., Duchien, L., Florin, G., Legond-Aubry, F., and Martelli, L. 2004. JAC: an aspect-based distributed dynamic framework. *Software Practice and Experience 34,* 12, 1119–1148.

Pierce, B. 2002. *Types and programming languages*. MIT press Cambridge, MA.

Schantz, R., Loyall, J., Atighetchi, M., and Pal, P. 2002. Packaging quality of service control behaviors for reuse. In *Object-Oriented Real-Time Distributed Computing, 2002. (ISORC 2002). Proceedings of Fifth IEEE International Symposium on*. 375–385.

Schmidt, D., Rohnert, H., Stal, M., and Schultz, D. 2000. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*. John Wiley & Sons, Inc. New York, NY, USA.

Seki, M. 2007. DRuby–A Distributed Object System for Ruby. `http://www.ruby-doc.org/stdlib/libdoc/drb/`.

Souder, T. and Mancoridis, S. 1999. A Tool for Securely Integrating Legacy Systems into a Distributed Environment. *Working Conference on Reverse Engineering*, 47–55.

The Object Management Group (OMG) 1997. *The Common Object Request Broker: Architecture and Specification*. The Object Management Group (OMG).

Thomas, D. and Hunt, A. 2001. *Programming Ruby*. Addison-Wesley Reading, MA.

Tilevich, E. and Smaragdakis, Y. 2002. J-Orchestra: Automatic Java Application Partitioning. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*.

Tilevich, E. and Smaragdakis, Y. 2008. NRMI: Natural and Efficient Middleware. *IEEE Transactions on Parallel and Distributed Systems*, 174–187.

Wohlstadter, E., Jackson, S., and Devanbu, P. 2003. DADO: enhancing middleware to support crosscutting features in distributed, heterogeneous systems. In *Proceedings of the International Conference on Software Engineering*. Vol. 186.

Wollrath, A., Riggs, R., and Waldo, J. 1996. A distributed object model for the javatm system. In *COOTS'96: Proceedings of the 2nd conference on USENIX Conference on Object-Oriented Technologies (COOTS)*. USENIX Association, Berkeley, CA, USA, 17–17.

Yang, C., Chen, C., Chang, Y., Chung, K., and Lee, J. 2006. Streaming support for Java RMI in distributed environments. *Proceedings of the 4th international symposium on Principles and practice of programming in Java*, 53–61.

Zhang, C. and Jacobsen, H. 2003. Refactoring middleware with aspects. *IEEE Transactions on Parallel and Distributed Systems 14,* 11, 1058–1073.

...